Monadic translation of intuitionistic sequent calculus

José Espírito Santo[1], Ralph Matthes[2], and Luís Pinto[1]
{jes,luis}@math.uminho.pt matthes@irit.fr

[1] Departamento de Matemática, Universidade do Minho, Portugal
[2] I.R.I.T. (C.N.R.S. and University of Toulouse III), France

February 20, 2009

Abstract. This paper proposes and analyses a monadic translation of an intuitionistic sequent calculus. The source of the translation is a typed λ-calculus previously introduced by the authors, corresponding to the intuitionistic fragment of the call-by-name variant of $\overline{\lambda}\mu\tilde{\mu}$ of Curien and Herbelin, and the target is a variant of Moggi's monadic meta-language, where the rewrite relation includes extra permutation rules that may be seen as variations of the "associativity" of bind (the Kleisli extension operation of the monad).
The main result is that the monadic translation simulates reduction strictly, so that strong normalisation (which is enjoyed at the target, as we show) can be lifted from the target to the source. A variant translation, obtained by adding an extra monad application in the translation of types, still enjoys strict simulation, while requiring one fewer extra permutation rule from the target.
Finally we instantiate, for the cases of the identity monad and the continuations monad, the meta-language into the simply-typed λ-calculus. By this means, we give a generic account of translations of sequent calculus into natural deduction, which encompasses the traditional mapping studied by Zucker and Pottinger, and CPS translations of intuitionistic sequent calculus.

1 Introduction

This paper is about a monadic translation of intuitionistic sequent calculus. By the latter we mean the intuitionistic, call-by-name fragment of Curien-Herbelin's system for classical logic [1]. In the spirit of the Curry-Howard correspondence, such a system is handled as an extension of the simply-typed λ-calculus, identified by the authors in [5], and named $\lambda\mathbf{J}^{\mathbf{m}se}$.

The target of the monadic translation is a variant of Moggi's monadic meta-language [12], named λ_{M} here. To recall, this is an extension of the simply-typed λ-calculus where the type system includes a monad M, and the term language includes constructions for the unit and the Kleisli extension (a. k. a. bind) operation of the monad. The main point is that the set of reduction rules of the meta-language is extended by two new rules, which can be seen as variations

of the usual "associativity" rule for bind, and which together with this "associativity" rule can be seen as forming a variation of one single principle in the ordinary λ-calculus, that we name assoc.

The monadic translation we introduce generalizes the ordinary monadic translation of the (call-by-name) λ-calculus [7], and, in particular, is based on the principle that functions from A to B are interpreted as functions from MA (computations of type A) to MB (computations of type B). The main result we obtain is a strict simulation theorem (one reduction step in the sequent calculus is mapped to one or more reduction steps in the monadic target). A variant of the monadic translation, based on the interpretation of functions from A to B as functions from MA to MMB, also enjoys strict simulation, and requires one less of the new reduction rules from the target system.

One of the uses of the above results is in obtaining *strong normalisation* for sequent calculus, i. e., the absence of an infinite sequence of proof transformations starting with a well-formed proof. Indeed, strong normalisation follows immediately from strict simulation, since the target system is itself strongly normalising. This fact, in turn, rests on the strong normalisability of the extension of λ-calculus with the assoc reduction rule [3]. This emphasis on strict simulation and strong normalisation follows the line of [5, 6], but is in contrast with the uses of the monadic language in the study of programming languages semantics and compilation, where other kinds of relationship between source and target calculi, like equational correspondence, or reflection, are often obtained [7, 16].

On the other hand, we may regard the monadic translations and their properties, not as a goal in itself, but as a parametric means to analyse a family of situations, via instantiation of the monad of the meta-language. In fact, we study two such instantiations, one for the identity monad, the other for the continuations monad, where by "instantiation" we mean composition of the monadic translation with an interpretation of the monadic language into the λ-calculus.

Through this method we obtain a *generic account* of translations of sequent calculus into natural deduction. The identity monad gives an analysis of what in our framework is the traditional mapping studied by Zucker and Pottinger [17, 14], together with some of its variants. The continuations monad obtains an analysis of a CPS translation of $\lambda\mathbf{J}^{\mathbf{m}se}$ similar to the one at the basis of [5].

The methodology of this generic account should be contrasted with that of [7]. There, it is the monadic translation that varies, in order to capture a family of situations (in the case of [7], several CPS translations), while the monad remains instantiated to the continuations monad. Here, the monadic translation remains fixed, while, by varying the monad, we uncover a common root to seemingly unrelated translations of sequent calculus into natural deduction.

The paper is organised as follows. Section 2 presents sequent calculus $\lambda\mathbf{J}^{\mathbf{m}se}$. Section 3 presents our version λ_{M} of the monadic meta-language. Section 4 defines and proves the properties of the monadic translation and its optimized variant, and strong normalisation for $\lambda\mathbf{J}^{\mathbf{m}se}$ is obtained. Section 5 gives the generic account of translation into natural deduction. Finally, Section 6 concludes with some remarks.

2 Intuitionistic Sequent Calculus

The calculus $\lambda\mathbf{J}^{\mathbf{m}se}$ that is used here has been proposed in [5] (whose journal version is [6]). It corresponds to the intuitionistic fragment of the call-by-name variant of $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [1]. We quite closely follow the presentation of the definition of $\lambda\mathbf{J}^{\mathbf{m}se}$ in [6].

There are three classes of expressions in $\lambda\mathbf{J}^{\mathbf{m}se}$:

$$\begin{array}{ll} \text{(Terms)} & t, u ::= x \mid \lambda x.t \mid \{c\} \\ \text{(Co-terms)} & l ::= [] \mid u :: l \mid (x)c \\ \text{(Commands)} & c ::= tl \end{array}$$

Terms can be variables (of which we assume a denumerable set ranged over by letters x, y, z), lambda-abstractions $\lambda x.t$ or coercions $\{c\}$ from commands to terms.

Co-terms provide means of forming lists of arguments, generalised arguments, or explicit substitutions. A co-term of the form $(x)c$, binds variable x in c and provides the generalised application facility. Operationally it can be thought of as "substitute for x in c". A co-term of the form $[]$ or $u :: l$ is called an *evaluation context* and is denoted by E. Evaluation contexts of the form $u :: l$ allow for multiary applications and, when passed to a term, indicate that, after consumption of argument u, computation should carry on with arguments in l. The co-term $[]$ marks the end of an evaluation context, while the expression $(x)x$ is just ill-formed and, in particular, not a co-term.

A command tl has a double role: if l is of the form $(x)c$, tl is an explicit substitution; otherwise, tl is a general form of application.

In writing expressions, sometimes we add parentheses to help their parsing. Also, we assume that the scope of binders λx and (x) extends as far as possible. We follow usual practise in that names of bound variables are considered as immaterial and that the binding occurrences on display are meant to be well-chosen so that no unwanted effects arise. It is then straightforward to define what it means to replace every free occurrence of variable x in a capture-avoiding way by a term t in a term u, co-term l or command c, yielding term $[t/x]u$, co-term $[t/x]l$ and command $[t/x]c$, respectively.

The calculus $\lambda\mathbf{J}^{\mathbf{m}se}$ has a form of sequent for each class of expressions:

$$\Gamma \vdash t : A \qquad \Gamma | l : A \vdash B \qquad \Gamma \xrightarrow{c} B$$

Letters A, B, C are used to range over the set of types (=formulas), built from a base set of type variables (ranged over by X) using the function type (that we write $A \supset B$). In sequents, contexts Γ are viewed as finite sets of declarations $x : A$, where no variable x occurs twice. The context $\Gamma, x : A$ is obtained from Γ by adding the declaration $x : A$, and will only be written if this yields again a valid context, i. e., if x is not declared in Γ. We can think of a term (resp. co-term) as an annotation for a selected formula in the *rhs* (resp. *lhs*). Commands annotate sequents generated as a result of logical cuts, where

Fig. 1. Typing rules of $\lambda\mathbf{J}^{\mathbf{m}se}$

$$\frac{}{\Gamma|[] : A \vdash A}\ LAx \qquad \frac{}{\Gamma, x : A \vdash x : A}\ RAx$$

$$\frac{\Gamma \vdash u : A \quad \Gamma|l : B \vdash C}{\Gamma|u :: l : A \supset B \vdash C}\ LIntro \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B}\ RIntro$$

$$\frac{\Gamma, x : A \xrightarrow{c} B}{\Gamma|(x)c : A \vdash B}\ LSel \qquad \frac{\Gamma \xrightarrow{c} A}{\Gamma \vdash \{c\} : A}\ RSel$$

$$\frac{\Gamma \vdash t : A \quad \Gamma|l : A \vdash B}{\Gamma \xrightarrow{tl} B}\ Cut$$

there is no selected formula on the *rhs* or *lhs*; as such we write them on top of the sequent arrow.

The typing rules of $\lambda\mathbf{J}^{\mathbf{m}se}$ are presented in Figure 1, stressing the parallel between left and right rules.

The standard typing rules for substitution for each syntactic class are admissible: replacing a variable of declared type A by a term of type A does not change the type. We also have the usual weakening rules: If a sequent with context Γ is derivable and Γ is replaced by a context Γ' that is a superset of Γ, then also this sequent is derivable.

We consider the following base reduction rules on expressions:[1]

$$\begin{array}{llll} (\beta) & (\lambda x.t)(u :: l) \to u((x)tl) & (\mu) & (x)xl \to l, \text{ if } x \notin l \\ (\pi) & \{tl\}E \to t\,(l@E) & (\epsilon) & \{t[]\} \to t \\ (\sigma) & t(x)c \to [t/x]c, & & \end{array}$$

where, in general, $l@l'$ is a co-term that represents an "eager" concatenation of l and l', viewed as lists, and is defined as follows[2]:

$$[]@l' = l' \qquad (u :: l)@l' = u :: (l@l') \qquad ((x)tl)@l' = (x)t\,(l@l')$$

Concatenation obeys to the following further admissible form of cut rule:

$$\frac{\Gamma|l : A \vdash B \quad \Gamma|l' : B \vdash C}{\Gamma|l@l' : A \vdash C}$$

[1] Naming practise for binding occurrences excludes x as a free variable in u or l in the left-hand side of rule β. The widening of the binding scope of x in the right-hand side is noteworthy, but it is only meant to correspond to weakening.

[2] Concatenation is "eager" in the sense that, in the last case, the right-hand side is not $(x)\{tl\}l'$ but, in the only important case that l' is an evaluation context E, its π-reduct. One immediately verifies $l@[] = l$ and $(l@l')@l'' = l@(l'@l'')$ by induction on l. Associativity would not hold with the lazy version of @.

The one-step reduction relation $\rightarrow$ is inductively defined as the term closure of the reduction rules.

For detailed comments on the reduction rules, the subject reduction property (that holds true), an analysis of normal forms and critical pairs (yielding local confluence) and the identification of $\lambda\mathbf{J}^{\mathbf{m}se}$ as the intuitionistic fragment of CBN $\overline{\lambda}\mu\tilde{\mu}$, see [5, 6].

We stress that the rule β does not execute any substitution. This makes a simulation of $\lambda\mathbf{J}^{\mathbf{m}se}$ in another system more difficult, not only because substitution is delayed, but also because the scope of the bound variable is enlarged.

3 Monadic Lambda-calculus

The main result of [5, 6] is a proof of strong normalization of $\lambda\mathbf{J}^{\mathbf{m}se}$ that does not refer to the strong normalization results by Lengrand [10] and Polonovski [13] about $\overline{\lambda}\mu\tilde{\mu}$, but by a syntactic transformation to simply-typed λ-calculus that strictly simulates reduction. The technique is a variation of continuation-passing style, called continuation-and-garbage-passing style [8]. CPS translations alone do not suffice for a strict simulation of all reductions. In the present article, we move from CPS translations to monadic translations, whose target we call monadic lambda-calculus. Strong normalisation of the monadic lambda-calculus itself does not rest any longer on simply-typed λ-calculus with only β-reduction; instead the following rule has to be added:

$$s((\lambda x.t)r) \rightarrow (\lambda x.st)r \ \ ,$$

where x is not free in s and s is a λ-abstraction. We call this rule assoc and the extension of λ-calculus obtained by adding it $\lambda[\beta, \mathsf{assoc}]$.

Proposition 1. *The calculus $\lambda[\beta, \mathsf{assoc}]$ is strongly normalizing, i. e., there is no infinite reduction sequence $t = t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$ with a typable term t.* [3]

Proof. A proof by Lengrand may be found in [11]. A stronger result was stated in [3], concerning the addition to the λ-calculus, not only of assoc (even without the abstraction proviso), but also of another permutation rule, due to Regnier [15], and named here perm. The "proof" of the strong result given in [3] was incomplete. A complete proof may be found in [4]. Strong normalisation of $\lambda[\beta, \mathsf{assoc}, \mathsf{perm}]$ will be needed below in Section 5.1 for translation F. □

Although our first aim is to give an alternative syntactic proof of strong normalization of $\lambda\mathbf{J}^{\mathbf{m}se}$, we want to be able to interpret $\lambda\mathbf{J}^{\mathbf{m}se}$ in as many monads as possible, and not just the identity monad. Hence, we take as target calculus the extension of simply-typed λ-calculus where the type system includes a monad—a type transformation called M as the single unary constant for building types—and the term language includes constructions for the unit and the Kleisli extension (a.k.a. bind) operation of the monad M, as follows: the term

[3] A term t is typable if there is a context Γ and a type A such that $\Gamma \vdash t : A$.

Fig. 2. Base reduction rules of λ_{M}

$$\begin{array}{rr}
(\beta_\lambda) & (\lambda x.t)s \to [s/x]t \\
(\beta_{\mathsf{bind}}) & \mathsf{bind}(\eta s, x.t) \to [s/x]t \\
(\pi_{\lambda,\lambda}) & (\lambda y.u)((\lambda x.t)r) \to (\lambda x.(\lambda y.u)t)r \\
(\pi_{\mathsf{bind},\lambda}) & \mathsf{bind}((\lambda x.t)r, y.u) \to (\lambda x.\mathsf{bind}(t, y.u))r \\
(\pi_{\mathsf{bind},\mathsf{bind}}) & \mathsf{bind}(\mathsf{bind}(r, x.t), y.u) \to \mathsf{bind}(r, x.\mathsf{bind}(t, y.u))
\end{array}$$

language is extended by the following clauses: If s is a term then ηs is a term, and if r and t are terms, then $\mathsf{bind}(r, x.t)$ is a term. The variable x is considered as bound by "$x.$" in t.

The usual typing rules of simply-typed λ-calculus are extended as follows:

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash \eta s : MA}\ \eta \qquad \frac{\Gamma \vdash r : MA \quad \Gamma, x : A \vdash t : MB}{\Gamma \vdash \mathsf{bind}(r, x.t) : MB}\ \mathsf{bind}$$

The monadic language was introduced by Moggi [12] as an equational theory and was used to interpret the computational lambda-calculus. Its corresponding reduction theory is considered in [7] and [16] and includes rules for the 3 monadic laws. Our monadic λ-calculus λ_{M} brings into play two more permutation rules. The base reduction rules of λ_{M} are shown in Figure 2. The implicit proviso for the three latter rules – the permutation rules – is that x is not free in $\lambda y.u$. Again, we write $\to$ for the term closure of the base reduction rules.

While β_{bind} and $\pi_{\mathsf{bind},\mathsf{bind}}$ correspond to two of the three monad laws, we do not need the eta rule of the monad $\mathsf{bind}(r, x.\eta x) \to r$.

Note that the rule $\pi_{\mathsf{bind},\lambda}$ orients the direct equational consequence of β_λ,

$$\mathsf{bind}((\lambda x.t)r, y.u) =_{\beta_\lambda} \mathsf{bind}([r/x]t, y.u) =_{\beta_\lambda} (\lambda x.\mathsf{bind}(t, y.u))r\ ,$$

in a specific way. Likewise, $\pi_{\lambda,\lambda}$ – which is just a different presentation of rule assoc – directs an equational consequence of β_λ. So, from a purely equational point of view, our notion of λ_{M} is not stronger than the ordinary one that only reflects the monad laws. Moreover, we even omitted the eta rule.

To the best of our knowledge, rules $(\pi_{\lambda,\lambda})$ and $(\pi_{\mathsf{bind},\lambda})$ have not been considered before in combination with the traditional monad rules. However as we show below, the enriched system λ_{M} enjoys good properties, which would hold even in presence of the monadic eta rule.

The λ_{M}-calculus can be interpreted in $\lambda[\beta, \mathsf{assoc}]$ so that strict simulation of reduction is obtained. The translation corresponds to defining the *identity monad* in $\lambda[\beta, \mathsf{assoc}]$. The translation $|\_| : \lambda_{\mathsf{M}} \to \lambda[\beta, \mathsf{assoc}]$ is defined on types by $|X| := X$, $|A \supset B| := |A| \supset |B|$ and $|MA| := A$, and is defined on terms by $|x| := x$, $|\lambda x.t| := \lambda x.|t|$, $|tu| := |t||u|$, $|\eta s| := |s|$ and $|\mathsf{bind}(r, x.t)| := (\lambda x.|t|)|r|$. Evidently, this respects the typing rules.

Lemma 1. *If $\Gamma \vdash s : A$ is derivable in λ_{M}, $|\Gamma| \vdash |s| : |A|$ is derivable in $\lambda[\beta, \mathsf{assoc}]$, where $|\Gamma|$ is the result of replacing each declaration $x : A$ in Γ by $x : |A|$.*

Under these definitions, β_λ and β_{bind} become β (the usual rule of λ-calculus that is β_λ, but quantified over a different set of terms), and all three permutation rules become the assoc rule. (The ordinary eta rule of the monad would be just mapped to one step of β.) Thus we have the strongest possible simulation result.[4]

Lemma 2. *If $t \to u$ in λ_{M}, $|t| \to |u|$ in $\lambda[\beta, \mathsf{assoc}]$.*

From the above result and strong normalization of $\lambda[\beta, \mathsf{assoc}]$, we immediately get the following result.

Corollary 1 *The calculus λ_{M} is strongly normalizing.*

Now, given that all critical pairs for the rules of λ_{M} are joinable, we also obtain a confluence result.

Corollary 2 *$\to$ is confluent for the typable terms of λ_{M}.*

4 Translations of $\lambda\mathbf{J}^{\mathbf{mse}}$ into Monadic λ-calculus

Here, we show how to translate $\lambda\mathbf{J}^{\mathbf{mse}}$ into λ_{M} such that one obtains strict simulation and thus can infer strong normalization of $\lambda\mathbf{J}^{\mathbf{mse}}$ from Corollary 1. Hence, this is an alternative syntactic proof of strong normalization of $\lambda\mathbf{J}^{\mathbf{mse}}$. While the translation in the following section works on the types in usual CBN fashion [7], a more complicated type translation in Section 4.2 even yields strict simulation within λ_{M} without the rule $\pi_{\lambda,\lambda}$.

4.1 Main monadic translation

A type A of $\lambda\mathbf{J}^{\mathbf{mse}}$ is translated to $\overline{A} = MA^*$ of λ_{M}, with the type A^* defined by recursion on A (where the definition of $\overline{A}$ is used as an abbreviation):

$$X^* = X \qquad \text{and} \qquad (A \supset B)^* = \overline{A} \supset \overline{B}$$

Note that, for the identity monad, this trivializes to $\overline{A} = A^* = A$. Any term t of $\lambda\mathbf{J}^{\mathbf{mse}}$ is translated into a term $\overline{t}$ of λ_{M}, any command c of $\lambda\mathbf{J}^{\mathbf{mse}}$ into a term $\overline{c}$ and any pair of a co-term l of $\lambda\mathbf{J}^{\mathbf{mse}}$ and a variable w of λ_{M}, with w not free in l, into a term l_w of λ_{M}.[5] This is done so that the typing rules in Figure 3 are derivable, where $\overline{\Gamma}$ is derived from Γ by replacing every $x : C$ in Γ by $x : \overline{C}$.

The definitions are in Figure 4, where it is understood that f, v and w are fresh variable names. The definition of $[]_w$ is given with the extra $(\lambda k.k)$ so as to form an (administrative) redex which will guarantee strict simulation of ϵ and of the initial cases of π, see the proofs of Lemma 4 and Theorem 1. Also $(\lambda v.l_v)(f\overline{u})$ is a redex for strict simulation purposes, and we will "monadically" abstract away from it in the optimized translation in Section 4.2.

[4] Strict simulation would just mean that one step in the source calculus is mapped to at least one step of the target calculus, which would be sufficient to inherit strong normalization of the source calculus from the target calculus.

[5] Whenever we write l_w (or E_w), it will be understood that w does not occur free in the co-term l (or E).

Fig. 3. Derived typing rules for monadic translation of $\lambda\mathbf{J}^{\mathbf{mse}}$

$$\frac{\Gamma \vdash t : A}{\overline{\Gamma} \vdash \overline{t} : \overline{A}} \qquad \frac{\Gamma \xrightarrow{c} A}{\overline{\Gamma} \vdash \overline{c} : \overline{A}} \qquad \frac{\Gamma | l : A \vdash B}{\overline{\Gamma}, w : \overline{A} \vdash l_w : \overline{B}}$$

Fig. 4. Monadic translation of $\lambda\mathbf{J}^{\mathbf{mse}}$

$$\begin{array}{lll} \overline{x} = x & []_w = (\lambda k.k)w & \\ \overline{\lambda x.t} = \eta(\lambda x.\overline{t}) & (u :: l)_w = \mathsf{bind}(w, f.(\lambda v.l_v)(f\overline{u})) & \overline{tl} = [\overline{t}/w]l_w \\ \overline{\{c\}} = \overline{c} & ((x)c)_w = (\lambda x.\overline{c})w & \end{array}$$

Lemma 3. *The translation satisfies* $\overline{[t/x]u} = [\overline{t}/x]\overline{u}$, $([t/x]l)_w = [\overline{t}/x](l_w)$ *and* $\overline{[t/x]c} = [\overline{t}/x]\overline{c}$. *The proviso for the second equation is that* x *is not* w. □

Lemma 4. *For* $w \notin E$[6] *and any* v, *one has* $[l_w/v]E_v \to^+ (l@E)_w$.

Proof. For $E = []$, we calculate

$$[l_w/v]([]_v) = [l_w/v]((\lambda k.k)v) = (\lambda k.k)l_w \to_{\beta\lambda} l_w = (l@[])_w \ .$$

For $E = u :: l'$, do induction on l.

Case []: $[(\lambda k.k)w/v]E_v \to_{\beta\lambda} [w/v]E_v = E_w$ (v once in E_v + renaming)

Case $u' :: l$:

$$\begin{array}{rcll} [(u' :: l)_w/v]E_v & = & \mathsf{bind}(\mathsf{bind}(w, g.(\lambda v'.l_{v'})(g\overline{u'})), f.(\lambda v.l'_v)(f\overline{u})) & \\ & \to_{\pi_{\mathsf{bind},\mathsf{bind}}} & \mathsf{bind}(w, g.\mathsf{bind}((\lambda v'.l_{v'})(g\overline{u'}), f.(\lambda v.l'_v)(f\overline{u}))) & \\ & \to_{\pi_{\mathsf{bind},\lambda}} & \mathsf{bind}(w, g.(\lambda v'.\mathsf{bind}(l_{v'}, f.(\lambda v.l'_v)(f\overline{u})))(g\overline{u'})) & \\ & = & \mathsf{bind}(w, g.(\lambda v'.[l_{v'}/w]E_w)(g\overline{u'})) & \\ & \to^+ & \mathsf{bind}(w, g.(\lambda v'.(l@E)_{v'})(g\overline{u'})) & \text{by IH for } l \\ & = & (u' :: (l@E))_w = ((u' :: l)@E)_w & \end{array}$$

Case $(y)c$ with $c = t_1 l_1$:

$$\begin{array}{rcll} [((y)c)_w/v]E_v & = & \mathsf{bind}((\lambda y.\overline{c})w, f.(\lambda v.l'_v)(f\overline{u})) & \\ & \to_{\pi_{\mathsf{bind},\lambda}} & (\lambda y.\mathsf{bind}(\overline{c}, f.(\lambda v.l'_v)(f\overline{u})))w & \\ & = & (\lambda y.[\overline{c}/v]E_v)w & \\ & = & (\lambda y.[[\overline{t_1}/v'](l_1)_{v'}/v]E_v)w & \\ & = & (\lambda y.[\overline{t_1}/v'][(l_1)_{v'}/v]E_v)w & \\ & \to^+ & (\lambda y.[\overline{t_1}/v'](l_1@E)_{v'})w & \text{by IH for } l_1 \\ & = & ((y)t_1(l_1@E))_w = (((y)c)@E)_w & \end{array}$$

□

[6] By writing $(l@E)_w$, we already implicitly assume that $w \notin E$, but this condition is not visible in the left-hand side of the statement, hence we indicate it.

Theorem 1 (Simulation) *If $t \to t'$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $\overline{t} \to^+ \overline{t'}$ in λ_{M}. If $l \to l'$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $l_w \to^+ l'_w$ in λ_{M}. If $c \to c'$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $\overline{c} \to^+ \overline{c'}$ in λ_{M}.*

Proof. We only have to consider a rewrite step at the root since the cases corresponding to the closure rules follow by routine induction. This is so because w has one free occurrence in l_w (it has only one occurrence), and so the definition of $\overline{tl}$ is uncritical ($\overline{t}$ cannot be lost as a subterm through substitution into l_w).

Case β: $(\lambda x.t)(u :: l) \to u(x)tl$.

$$\begin{array}{rcl}\overline{(\lambda x.t)(u :: l)} & = & \mathsf{bind}(\eta(\lambda x.\overline{t}), f.(\lambda v.l_v)(f\overline{u})) \\ & \to_{\beta_{\mathsf{bind}}} & (\lambda v.l_v)((\lambda x.\overline{t})\overline{u}) \\ & \to_{\pi_{\lambda,\lambda}} & (\lambda x.(\lambda v.l_v)\overline{t})\overline{u} \\ & \to_{\beta_\lambda} & (\lambda x.[\overline{t}/v]l_v)\overline{u} = (\lambda x.\overline{tl})\overline{u} = \overline{u(x)tl}\end{array}$$

Case σ: $t(x)c \to [t/x]c$:

$$\overline{t(x)c} = (\lambda x.\overline{c})\overline{t} \to_{\beta_\lambda} [\overline{t}/x]\overline{c} = \overline{[t/x]c} \quad \text{(Lemma 3)}$$

Case ϵ: $\{t[]\} \to t$: $\quad \overline{\{t[]\}} = (\lambda k.k)\overline{t} \to_{\beta_\lambda} \overline{t}$

Case μ: $(x)xl \to l$, if $x \notin l$.

$$((x)xl)_w = (\lambda x.\overline{xl})w = (\lambda x.[x/w]l_w)w \to_{\beta_\lambda} [w/x][x/w]l_w = l_w$$

Case π: $\{tl\}E \to t\,(l@E)$. Apply substitution $[\overline{t}/w]$ to Lemma 4:

$$\overline{\{tl\}E} = [\overline{tl}/v]E_v = [[\overline{t}/w]l_w/v]E_v = [\overline{t}/w][l_w/v]E_v \to^+ [\overline{t}/w](l@E)_w = \overline{t(l@E)}\ ,$$

using the usual lemma about substitution interchange in the third step. □

Corollary 3 *The calculus $\lambda\mathbf{J}^{\mathbf{m}se}$ is strongly normalizing.*

Proof. Use the previous theorem, the preservation of typability expressed in Figure 3 and Corollary 1.

We remark that $\pi_{\lambda,\lambda}$ would not have been necessary if rule β of $\lambda\mathbf{J}^{\mathbf{m}se}$ were already σ-reduced on the right-hand side, thus with $[u/x]tl$. The calculation would be as follows:

$$\begin{array}{rcll}\overline{(\lambda x.t)(u :: l)} & = & \mathsf{bind}(\eta(\lambda x.\overline{t}), f.(\lambda v.l_v)(f\overline{u})) & \\ & \to_{\beta_{\mathsf{bind}}} & (\lambda v.l_v)((\lambda x.\overline{t})\overline{u}) & \\ & \to_{\beta_\lambda} & (\lambda v.l_v)([\overline{u}/x]\overline{t}) & \\ & = & (\lambda v.l_v)\overline{[u/x]t} & \text{(Lemma 3)} \\ & \to_{\beta_\lambda} & [\overline{[u/x]t}/v]l_v = \overline{[u/x]tl} & \end{array}$$

Our monadic translation when restricted to λ-calculus essentially captures the usual CBN monadic translation [7], call it $(\_)^\circ$. This translation for variables and λ-abstraction behaves as our translation, and for applications does $(tu)^\circ := \mathsf{bind}(t^\circ, f.fu^\circ)$. Our translation of a λ-calculus application tu, encoded in $\lambda\mathbf{J}^{\mathbf{m}se}$ as $t(u :: [])$, reaches the expected term after two β_λ-steps:

$$\overline{t(u :: [])} = \mathsf{bind}(\overline{t}, f.(\lambda v.[]_v)(f\overline{u})) \to^2_{\beta_\lambda} \mathsf{bind}(\overline{t}, f.f\overline{u})$$

We also notice that the property "$t \to_\beta u$ in the λ-calculus $\Rightarrow t^\circ \to_{\beta_{\mathsf{bind}},\beta_\lambda} u^\circ$ in the λ_{M}-calculus", that holds of mapping $(\_)^\circ$ (an easy, perhaps new result), is also shared by our translation.

Fig. 5. Optimized monadic translation of $\lambda\mathbf{J}^{\mathbf{m}se}$

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\overline{x} & = & x & []_w & = & (\lambda k.k)w & & & \\
\overline{\lambda x.t} & = & \eta(\lambda x.\eta\overline{t}) & (u :: l)_w & = & \mathsf{bind}(w, f.\mathsf{bind}(f\overline{u}, v.l_v)) & \overline{tl} & = & [\overline{t}/w]l_w \\
\overline{\{c\}} & = & \overline{c} & ((x)c)_w & = & (\lambda x.\overline{c})w & & &
\end{array}$$

4.2 Optimized translation

Now, a translation is given that allows simulation of $\lambda\mathbf{J}^{\mathbf{m}se}$ even in λ^{-}_{M} that is obtained from λ_{M} by omitting the rule $\pi_{\lambda,\lambda}$. The symbols of the previous subsection will be reused, but their definition will be changed.

A type A of $\lambda\mathbf{J}^{\mathbf{m}se}$ is translated to $\overline{A} = MA^*$ of λ_{M}, with the type A^* defined by recursion on A (where the definition of $\overline{A}$ is used as an abbreviation):

$$X^* = X \qquad \text{and} \qquad (A \supset B)^* = \overline{A} \supset M\overline{B}$$

Note that, for the identity monad, this again trivializes to $\overline{A} = A^* = A$. But the crucial change is that an extra M is inserted on top of $\overline{B}$ in the translation of $A \supset B$. For the special case of $MA = \neg\neg A$, this is logically equivalent to the translation used in [5, 6].

Any term t of $\lambda\mathbf{J}^{\mathbf{m}se}$ is translated into a term $\overline{t}$ of λ_{M}, any command c of $\lambda\mathbf{J}^{\mathbf{m}se}$ into a term $\overline{c}$ and any pair of a co-term l of $\lambda\mathbf{J}^{\mathbf{m}se}$ and a variable w of λ_{M}, with w not free in l, into a term l_w of λ_{M}. This is done so that the typing rules in Figure 3 are again derivable, where, obviously, all symbols have to be interpreted according to the current definitions.

The definitions are in Figure 5, where the usual freshness assumptions are understood. Changes with respect to Figure 4 concern λ-abstraction with an extra η and $(u :: l)_w$ where bind replaces the β redex. In fact, $\mathsf{bind}(f\overline{u}, v.l_v)$ is just the monadic version of $(\lambda v.l_v)(f\overline{u})$ that was used formerly. For the identity monad, the translation thus agrees with that of Section 4.1. However, in the general case, $\mathsf{bind}(f\overline{u}, v.l_v)$ would not be well-typed with the definitions of Section 4.1. For $\Gamma|u :: l : A \supset B \vdash C$, one would have $w : \overline{A \supset B}$ and hence $f : (A \supset B)^*$. Therefore, $f\overline{u}$ would have type $\overline{B}$ and finally $v : B^*$, which is not enough. We remark that one can base an alternative translation with A^* as in Section 4.1 on the idea of enforcing the admissible rule $\Gamma|l : A \vdash B \Rightarrow \overline{\Gamma}, w : A^* \vdash l_w : \overline{B}$. Simulation results for this alternative translation needed extensions of the η rule $\mathsf{bind}(t, x.\eta x) \to t$ that did not seem to be well justified.

Lemma 3 also holds for the definitions of the present section.

Theorem 2 (Simulation) *If $t \to t'$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $\overline{t} \to^+ \overline{t'}$ in λ^{-}_{M}. If $l \to l'$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $l_w \to^+ l'_w$ in λ^{-}_{M}. If $c \to c'$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $\overline{c} \to^+ \overline{c'}$ in λ^{-}_{M}.*

Proof. As in the proof of Theorem 1, it suffices to consider the base cases of reduction at the root. The cases σ, ϵ and μ can be copied verbatim from the proof of Theorem 1. For β, one calculates that

$$\overline{(\lambda x.t)(u :: l)} \to_{\beta_{\mathsf{bind}}} \mathsf{bind}((\lambda x.\eta\overline{t})\overline{u}, v.l_v) \to_{\pi_{\mathsf{bind},\lambda}} (\lambda x.\mathsf{bind}(\eta\overline{t}, v.l_v))\overline{u} \to_{\beta_{\mathsf{bind}}} \overline{u(x)tl}$$

Case π: $\{tl\}E \to t\,(l@E)$. The treatment of $E = []$ is immediate due to the extra redex in the definition of $[]_w$.

Sub-case $E = u :: l'$. We have to show $\overline{\{tl\}E} \to^+ \overline{t(l@E)}$, which is done by induction on l, simultaneously for all t. □

5 Generic Account of Translation into Natural Deduction

An instantiation of the monadic translation with a particular monad gives an interpretation of the intuitionistic sequent calculus $\lambda\mathbf{J}^{\mathbf{m}se}$ into natural deduction. In this section we show that two such instantiations relate to known interpretations, namely variants of both the Zucker-Pottinger translation and a CPS translation. These interpretations receive, thus, a generic account through the monadic translation.

5.1 Direct Translations

In this subsection we study certain "direct" translations of $\lambda\mathbf{J}^{\mathbf{m}se}$ into the λ-calculus. One of these, named N here, implements the traditional interpretation of sequent calculus into natural deduction studied by Zucker [17] and Pottinger [14]. The directness comes from the fact no translation of types is involved, and also because these translations give a straightforward expression in terms of the λ-calculus of the computational interpretations of $\lambda\mathbf{J}^{\mathbf{m}se}$-expressions. The direct translations, as we will see, turn out to be related to the monadic translation, when the latter is instantiated with the identity monad.

A direct translation. Let F be the mapping from $\lambda\mathbf{J}^{\mathbf{m}se}$ to λ, based on the idea of mapping, say, $t(u_1 :: u_2 :: [])$ and $t(u_1 :: u_2 :: (x)c)$ to

$$(\lambda x.x)(rs_1s_2) \text{ and } (\lambda x.s)(rs_1s_2)\ ,$$

where r, s_i, and s are the translations of t, u_i, and c, respectively. Formally, F is given by

$$\begin{array}{ccccc} F(x) = x & & F(r,[]) = (\lambda x.x)r & & \\ F(\lambda x.t) = \lambda x.F(t) & & F(r, u :: l) = F(rF(u), l) & & F(tl) = F(F(t), l) \\ F(\{c\}) = F(c) & & F(r,(x)c) = (\lambda x.F(c))r & & \end{array}$$

We will need the target of F to be equipped not only with the assoc reduction rule, but also with

$$(\lambda x.t)rs \to (\lambda x.ts)r\ ,$$

for x not free in s (a proviso that, as for assoc, already follows from the variable convention). This is a well-known permutation rule [15, 9], which we name here perm. Let $\lambda[\beta, \mathsf{assoc}, \mathsf{perm}]$ be the λ-calculus equipped with both assoc and perm.

As mentioned in the proof of Proposition 1, normalisation of $\lambda[\beta, \mathsf{assoc}, \mathsf{perm}]$ holds as a consequence of a result stated in [3] and fully proved in [4].

Proposition 2. *If $t \to u$ in $\lambda\mathbf{J}^{\mathbf{m}se}$ then $F(t) \to^+ F(u)$ in $\lambda[\beta, \mathsf{assoc}, \mathsf{perm}]$.*

Proof. A by-hand proof would be possible, but we give an indirect proof, joining scattered results from the literature. The point is that F is the composition of the following mappings

$$\lambda\mathbf{J}^{\mathbf{m}se} \xrightarrow{(\_)^\circ} \lambda\mathbf{J}^{\mathbf{m}s} \subset \lambda^{\mathbf{G}tz} \xrightarrow{(\_)^*} \lambda\mathbf{s} \xrightarrow{(\_)^\sharp} \lambda[\beta\pi]$$

where $(\_)^\circ : \lambda\mathbf{J}^{\mathbf{m}se} \to \lambda\mathbf{J}^{\mathbf{m}s}$ comes from [6], $(\_)^* : \lambda^{\mathbf{G}tz} \to \lambda\mathbf{s}$ comes from [2], and $(\_)^\sharp : \lambda\mathbf{s} \to \lambda[\beta\pi]$ comes from [3]. $\lambda\mathbf{J}^{\mathbf{m}s}$ is the system preceding $\lambda\mathbf{J}^{\mathbf{m}se}$ in the "spectrum" of intuitionistic systems studied in [5, 6]. The difference relatively to $\lambda\mathbf{J}^{\mathbf{m}se}$ is that there is neither a separate class of commands, nor co-terms $[]$; instead, selection has the general form $(x)t$. $\lambda^{\mathbf{G}tz}$ is identical to $\lambda\mathbf{J}^{\mathbf{m}s}$, except that it has a more general π reduction rule, in that the call-by-name restriction is not imposed, and the concatenation operator is lazy; so each π step in $\lambda\mathbf{J}^{\mathbf{m}s}$ corresponds to one or more π steps in $\lambda^{\mathbf{G}tz}$. $\lambda\mathbf{s}$ is λ plus a substitution construction, equipped with rules for generating (β), executing (σ), and delaying (π) substitution. $\lambda[\beta\pi]$ is identical to $\lambda[\beta, \mathsf{assoc}, \mathsf{perm}]$, except that in $\lambda[\beta\pi]$ the abstraction proviso in the assoc rule is not imposed. [7] Mapping $(\_)^\circ$ erases the coercion $\{-\}$ and encodes $[]$ as $(x)x$. Mapping $(\_)^*$ has the same spirit as F, except that tl is mapped to a substitution, instead of a β-redex. Finally mapping $(\_)^\sharp$ "raises" substitutions to β-redexes. The present proposition is corollary of three simulation results: Proposition 3.6 of [6] concerning $(\_)^\circ$, Proposition 1 of [2] concerning $(\_)^*$, and Proposition 7 of [3] concerning $(\_)^\sharp$. All three state that each reduction step of the source generates one or more reduction steps of the target, except in one case: $(\_)^\sharp$ collapses β steps of $\lambda\mathbf{s}$. So, one has to supplement Proposition 1 of [2] with the remark - useless for the purposes of [2], but needed now - that $(\_)^*$ always generates at least one reduction step different from β in the target, when translating a reduction step of its source. Finally we observe that the simulation property of $(\_)^\sharp$ still holds when one takes the assoc rule of $\lambda[\beta\pi]$ with the abstraction proviso, and therefore the target of $(\_)^\sharp$ can be taken as $\lambda[\beta, \mathsf{assoc}, \mathsf{perm}]$. □

Identity-monadic translations. Let G be the composition of the monadic translation with the mapping $|\_| : \lambda_{\mathsf{M}} \to \lambda[\beta, \mathsf{assoc}]$ from the end of Section 3. G maps, say, $t(u_1 :: u_2 :: [])$ and $t(u_1 :: u_2 :: (x)c)$ respectively to

$$(\lambda f.(\lambda z.(\lambda f'.(\lambda z'.(\lambda x.x)z)(f's_2))z')(fs_1))r$$
$$(\lambda f.(\lambda z.(\lambda f'.(\lambda z'.(\lambda x.s)z)(f's_2))z')(fs_1))r$$

if we let again r, s_i, and s be the translations of t, u_i, and c, respectively. A recursive definition of G is:

$$\begin{array}{lll} G(x) = x & []_w = (\lambda k.k)w & \\ G(\lambda x.t) = \lambda x.G(t) & (u :: l)_w = (\lambda f.(\lambda v.l_v)(fG(u)))w & G(tl) = [G(t)/w]l_w \\ G(\{c\}) = G(c) & ((x)c)_w = (\lambda x.G(c))w & \end{array}$$

[7] The idea is that perm and the relaxed assoc (called π_1 and π_2 in [3] respectively) form a coherent set of rules for "delaying" a "substitution" $(\lambda x.-)r$ surrounding a term t, whenever this t occurs in the function or argument positions of an application.

Proposition 3. *If $t \to u$ in $\lambda\mathbf{J}^{\mathbf{m}se}$ then $G(t) \to^+ G(u)$ in $\lambda[\beta, \mathsf{assoc}]$.*

Proof. Immediate consequence of Theorem 2, and the 1-1 mapping of reduction steps given by $|\_|$.

Comparison of translations. The previous proposition guarantees that, by changing from the encoding of commands of F to the encoding of commands of G, we dispense with perm in the target. For instance, consider

$$c_1 = \{t(u_1 :: (x)c)\}(u_1' :: (y)c') \qquad \text{and} \qquad c_2 = t(u_1 :: (x)v(u_1' :: (y)c'))$$

with $c = v[]$. Then $s = F(c) = (\lambda z.z)F(v)$ and $c_1 \to_\pi c_2$ in $\lambda\mathbf{J}^{\mathbf{m}se}$. In the target we have (if $s_1 = F(u_1)$, $s' = F(c')$, and $s_1' = F(u_1')$)

$$F(c_1) = \big(\lambda y.s'\big)\big((\lambda x.s)(rs_1)s_1'\big) \qquad \text{and} \qquad F(c_2) = \big(\lambda x.(\lambda y.s')(F(v)s_1')\big)\big(rs_1\big)$$

After reducing s to $F(v)$ and performing one perm step, one obtains from $F(c_1)$ the term $(\lambda y.s')((\lambda x.F(v)s_1')(rs_1))$, which in turn reaches $F(c_2)$ after one assoc step. On the other hand,

$$G(c_1) = \Big(\lambda f'.(\lambda z'.(\lambda y.s')z')(f's_1')\Big)\Big((\lambda f.(\lambda z.(\lambda x.s)z)(fs_1))r\Big)$$
$$G(c_2) = \Big(\lambda f.\big((\lambda z.(\lambda x.(\lambda f'.(\lambda z'.(\lambda y.s')z')(f's_1'))G(v))z\big)\big(fs_1\big)\Big)r$$

with $s = G(c) = (\lambda z.z)G(v)$, $s_1 = G(u_1)$, $s' = G(c')$, and $s_1' = G(u_1')$. Now $G(c_1)$ reaches $G(c_2)$ after 3 assoc steps, provided s is first reduced to $G(v)$.

The proximity between F and G becomes clearer if we give the definition of F in the following style:

$$\begin{array}{ccccc} F(x) = x & & []_w = (\lambda k.k)w & & \\ F(\lambda x.t) = \lambda x.F(t) & & (u :: l)_w = [wF(u)/v]l_v & & F(tl) = [F(t)/w]l_w \\ F(\{c\}) = F(c) & & ((x)c)_w = (\lambda x.F(c))w & & \end{array}$$

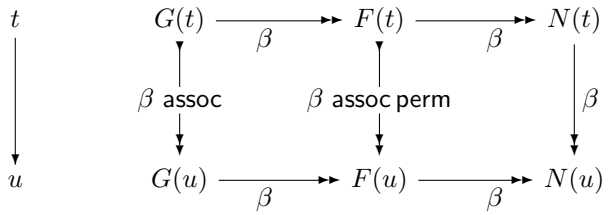
So, the only difference between F and G is in the clause for $(u :: l)_w$, where the two β-redexes appearing in the clause for G are contracted in the clause for F.

F translates the left introduction $u :: l$ as the traditional map between sequent calculus and natural deduction does - through a combination of application and substitution; but, because of the definitions of $[]_w$ and $((x)c)_w$, F's translation of a cut generates a β-redex whose contractum (a certain substitution) would be the translation of that cut by the traditional mapping. So, if we let N denote the traditional mapping, N is defined as F, except that now we put $[]_w = w$ and $((x)c)_w = [w/x]N(c)$. Notice that N also corresponds to taking the definition of G and uniformly contracting all β-redexes in the clauses defining l_w.

Proposition 4. *If $t \to u$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $Nt \to_\beta^* Nu$ in the λ-calculus.*[8]

[8] In fact, all but β-steps are identified by N, meaning that only the reduction rule corresponding to the key step of cut-elimination has a non-trivial translation. This

Let us sum up in the following diagram, where double-headed arrows denote 0 or more steps of reduction, except for the two central, vertical arrows, which denote 1 or more steps (a fact signaled by a little black triangle).



The map G, obtained by instantiating the monadic map, has a sharper simulation property than the previously known maps.

5.2 CPS Translation

In this subsection we introduce a CPS translation of $\lambda\mathbf{J}^{\mathbf{m}se}$, and compare it with the monadic translation instantiated with the continuations monad.

A CPS translation. Let A^* be given by $X^* = X$ and $(A \supset B)^* = \widehat{A} \supset \widehat{B}$, where $\widehat{A} = \neg\neg A^*$. The CPS translation of $t \in \lambda\mathbf{J}^{\mathbf{m}se}$ - denoted $\widehat{t}$ - is defined as $\lambda k.(t : k)$, where the so-called colon-operation is given as follows:

$$\begin{array}{rlrl} (x : K) &= xK & ([] : K) &= \lambda w.wK \\ (\lambda x.t : K) &= K(\lambda x.\widehat{t}) & (u :: l : K) &= \lambda w.w(\lambda f.(l : K)(f\widehat{u})) \\ (\{c\} : K) &= (c : K) & ((x)c : K) &= \lambda x.(c : K) \end{array}$$

$$\begin{array}{rl} (t[] : K) &= (t : K) \\ (t(u :: l) : K) &= (t : \lambda f.(l : K)(f\widehat{u})) \\ (t(x)c : K) &= ((x)c : K)\widehat{t} \end{array}$$

This CPS translation is considered in [6]. In [5, 6] a different CPS translation is given, based on the definition $(A \supset B)^* = \neg\widehat{B} \supset \neg\widehat{A}$, and a weak simulation result for it is proved, stating that each reduction step in the source $\lambda\mathbf{J}^{\mathbf{m}se}$ is mapped to 0 or more β-reduction steps in the λ-calculus. A variant of the proof sketched in [5, 6] gives an even weaker result for the present CPS translation.

Proposition 5. *If $t \to u$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $\widehat{t} \to^* \widehat{u}$ in $\lambda[\beta, \mathsf{assoc}]$.*

Indeed one needs assoc in the target, precisely for the simulation of β:

$$((\lambda x.t)(u :: []) : K) \quad = \quad (\lambda f.([] : K)(f\widehat{u}))(\lambda x.\widehat{t})$$

agrees with the properties of the map $(\_)^\flat : LJ \to \lambda$ studied in [3], a map implementing the traditional translation of another sequent calculus, named LJ there. Each β-step in the source of N is guaranteed to generate exactly one step in the target only when it happens at root position. The same applies to $(\_)^\flat$, but is not acknowledged in Proposition 10 of [3].

$$\begin{array}{rl} \to_\beta & ([] : K)((\lambda x.\widehat{t})\widehat{u}) \\ \to_{\mathsf{assoc}} & (\lambda x.([] : K)\widehat{t})\widehat{u} \\ \to_\beta^2 & (\lambda x.(t : K))\widehat{u} = (u(x)(t[]) : K) \end{array}$$

For this special case of β, the LHS also reduces to the RHS using β and perm. However, it is no longer possible to replace assoc by perm if instead of the empty list we have another list format.

Despite its weaker simulation properties, the CPS translation satisfying $(A \supset B)^* = \widehat{A} \supset \widehat{B}$ that we consider here is better suited for the purpose of comparing with the monadic translation.

Continuation-monadic translation. We define the *continuations monad* in the λ-calculus. Let $MA := \neg\neg A$, so that $\overline{A} = \widehat{A}$. Put

$$\eta t := \lambda k.kt \qquad \text{and} \qquad \mathsf{bind}(r, x.s) := \lambda k.r(\lambda x.sk)$$

We may see these definitions of M, η, and bind as giving an interpretation of λ_{M} into λ. Under this interpretation, the reduction rules of λ_{M} hold as $\beta\eta$-equalities in the λ-calculus.

The instantiation of the monadic translation (that is, the composition of the monadic translation with the present interpretation of λ_{M} into λ) gives:

$$\begin{array}{rlrl} C(x) &= x & []_w &= (\lambda k.k)w \\ C(\lambda x.t) &= \lambda k.k(\lambda x.C(t)) & (u :: l)_w &= \lambda k.w\big(\lambda f.(\lambda v.l_v)(fC(u))k\big) \\ C(\{c\}) &= C(c) & ((x)c)_w &= (\lambda x.C(c))w \end{array}$$

$$C(tl) = [C(t)/w]l_w$$

One immediately obtains that C maps each reduction step in $\lambda\mathbf{J}^{\mathbf{ms}}$ to a $\beta\eta$-equality in the λ-calculus.

Comparison with CPS. $C(t)$ is close to the CPS translation $\widehat{t}$. To see this, we introduce the "colon-free" translation $\widetilde{t}$, an intermediate point between $C(t)$ and $\widehat{t}$. $\widetilde{t}$ is defined as $\lambda k.(t : -)k$, where $(t : -)$ is given by:

$$\begin{array}{rlrlrl} (x : -) &= x & []^w &= w & & \\ (\lambda x.t : -) &= \lambda k.k(\lambda x.\widetilde{t}) & (u :: l)^w &= \lambda k.w(\lambda f.(\lambda v.l^v k)(f\widetilde{u})) & (tl : -) &= [\widetilde{t}/w]l^w \\ (\{c\} : -) &= (c : -) & ((x)c)^w &= \lambda k.(\lambda x.(c : -)k)w & & \end{array}$$

Then one proves: i) $(t : -)K$ reduces to $(t : K)$ - whence $\widetilde{t}$ reduces to $\widehat{t}$; ii) $(c : -)K$ reduces to $(c : K)$; iii) $\lambda w.l^w K$ reduces to $(l : K)$. The proof is a simultaneous induction on t, c, and l. Here reduction means 0 or more *administrative* β-steps. An administrative step is of one of two forms:

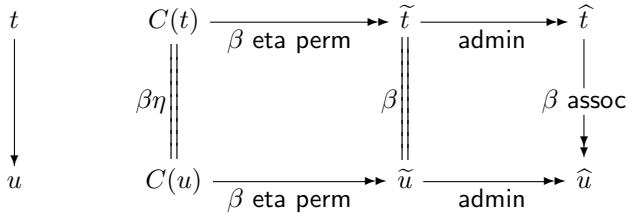
1. reduction of redexes $(\lambda k.t)K$ - pushing continuations inside;
2. $\lambda w.(\lambda x.t)w \to_\beta \lambda x.t$, with w not in t - notice the implicit α-conversion.

If $t \to u$ in $\lambda\mathbf{J}^{\mathbf{mse}}$, then $\widetilde{t} =_\beta \widetilde{u}$. This follows from the remarks just made, together with Proposition 5 and the fact $\mathsf{assoc} \subseteq =_\beta$.

After the transfiguration of the CPS translations, it is perspicuous that: i) $C(t)$ reduces to $(t : -)$ (which in turn η-expands to $\widetilde{t}$; we let $\mathsf{eta} = \eta^{-1}$ and refer to η-expansion as eta-reduction); ii) $C(c)$ reduces to $(c : -)$; iii) l_w reduces to l^w. The proof is again a simultaneous induction on t, c, and l. Here reduction means 0 or more steps of one of the following forms:

1. β, for reducing the redex $[]_w$;
2. eta, needed to bridge $(t : -)$ and $\widetilde{t}$;
3. eta followed by perm, for reducing the generic $(\lambda x.C(c))w$ to the continuations-monad-specific $\lambda k.(\lambda x.C(c)k)w$.
4. perm for bridging the change in the placement of variable k, when moving from the clause for $(u :: l)_w$ to the clause for $(u :: l)^w$.

Let us sum up with this diagram.



These results show how close is the CPS translation $\widehat{t}$ of being a mere instantiation of our monadic translation; and how the CPS translation helps explaining, in terms of reduction, the equational relationship existing between $C(t)$ and $C(u)$, when t and u are related by a reduction step in the source calculus.

6 Final remarks

This paper raises two issues that deserve further consideration. The first issue is whether the technique of "garbage-passing", as used in the translation of λ-calculus with control operators in [8] and later for translation of intuitionistic sequent calculus [5, 6], can be captured through some monad. Less ambitiously, one would hope for a precise comparison that allows to see why there is no need for extra rules such as assoc in the target of the garbage-passing translation. The second issue is the systematization of "associativity" principles in the monadic meta-language; indeed, it is conspicuous that one principle is missing, namely $\pi_{\lambda,\mathsf{bind}}$, which reads $(\lambda y.u)\mathsf{bind}(r, x.t) \to \mathsf{bind}(r, x.(\lambda y.u)t)$. The uses and properties of this rule are not yet entirely clear.

Acknowledgements: We are thankful to the referees for their constructive feedback. The first and third authors are supported by FCT through the Centro de Matemática da Universidade do Minho. The second author thanks for an invitation by that institution to Braga in May 2008. All authors were supported by the European Union FP6-2002-IST-C Coordination Action 510996 "Types for Proofs and Programs" and the first and third authors are also supported by RESCUE FCT project PTDC/EIA/65862/2006.

References

1. P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of 5th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP '00), Montréal*, pages 233–243. IEEE, 2000.
2. J. Espírito Santo. Completing Herbelin's programme. In S. Ronchi Della Rocca, editor, *Proceedings of TLCA'07*, volume 4583 of *LNCS*, pages 118–132. Springer-Verlag, 2007.
3. J. Espírito Santo. Delayed substitutions. In Franz Baader, editor, *Proceedings of RTA'07*, LNCS, pages 169–183. Springer-Verlag, 2007.
4. J. Espírito Santo. Addenda to "Delayed Substitutions", 2008 (Manuscript available from the author's web page).
5. José Espírito Santo, Ralph Matthes, and Luís Pinto. Continuation-passing style and strong normalisation for intuitionistic sequent calculi. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications (TLCA) 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 133–147. Springer Verlag, 2007.
6. José Espírito Santo, Ralph Matthes, and Luís Pinto. Continuation-passing style and strong normalisation for intuitionistic sequent calculi. *Logical Methods in Computer Science*, 2009. To appear.
7. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471. ACM, 1994.
8. Satoshi Ikeda and Koji Nakazawa. Strong normalization proofs by CPS-translations. *Information Processing Letters*, 99:163–170, 2006.
9. A.J. Kfoury and J.B. Wells. New notions of reduction and non-semantic proofs of beta-strong normalisation in typed lambda-calculi. In *Proceedings of LICS'95*, pages 311–321, 1995.
10. S. Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In B. Gramlich and S. Lucas, editors, *Post-proc. of the 3rd Workshop on Reduction Strategies in Rewriting and Programming (WRS'03)*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.
11. S. Lengrand. Temination of lambda-calculus with the extra call-by-value rule known as assoc. *arXiv:0806.4859v2*, 2007.
12. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
13. E. Polonovski. Strong normalization of $\overline{\lambda}\mu\tilde{\mu}$ with explicit substitutions. In Igor Walukiewicz, editor, *Proc. of 7th Int. Conference on Foundations of Software Sciences and Computation Structures (FoSSaCS 2004)*, volume 2987 of *Lecture Notes in Computer Science*, pages 423–437. Springer-Verlag, 2004.
14. G. Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12(3):323–357, 1977.
15. Laurent Regnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 126(2):281–292, 1994.
16. Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.
17. J. Zucker. The correspondence between cut-elimination and normalization. *Annals of Mathematical Logic*, 7(1):1–112, 1974.